CHAROTAR UNIVERSITY OF SCIENCE AND TECHNOLOGY

CE251: PROGRAMMING IN JAVA

Streams and Input/Output File

Ms. Sachi Joshi
Assistant Professor

Devang Patel Institute of Advance Technology and Research

Streams & Files

The objectives of this chapter are:

- To understand the principles of I/O streams and where to use them
- To understand the options and limitations of I/O streams
- To become comfortable with the mechanisms for accessing the file system

Introduction

- So far we have used variables and arrays for storing data inside the programs.

- This approach poses the following limitations:

Limitation

- The data is lost when variable goes out of scope or when the program terminates. That is data is stored in temporary/main memory is released when program terminates.
- It is difficult to handle large volumes of data.

Solution

- We can overcome this problem by storing data on secondary storage devices such as floppy or hard disks.
- The data is stored in these devices using the concept of Files and such data is often called persistent data.

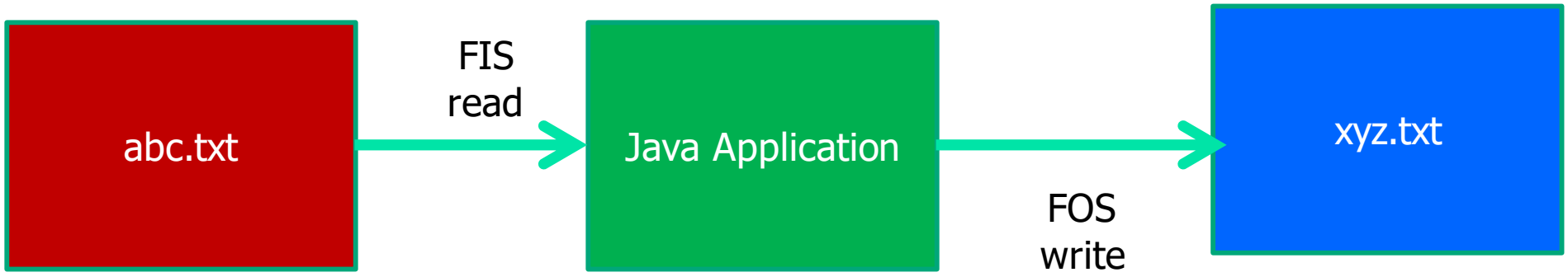
File Processing

- Storing and manipulating data using files is known as file processing.
- Reading/Writing of data in a file can be performed at the level of bytes, characters, or fields depending on application requirements.
- Java also provides capabilities to read and write class objects directly. The process of reading and writing objects is called object serialisation.

C Input/Output Revision

```
FILE* fp;

fp = fopen("In.file", "rw");
fscanf(fp, ……);
frpintf(fp, …..);
fread(……., fp);
fwrite(……., fp);
```

Java Application Requirement



Lets write a code

```
import java.io.*;
class  FileIOExample
{
        public static void main(String[] args)
        {
                FileInputStream fis = new FileInputStream("abc.txt");
                FileOutputStream fos = new FileOutputStream("xyz.txt");
                int c;
                while((c=fis.read())!=-1)
                {
                        fos.write(c);
                }
                fis.close();
                fos.close();
        }
}
```

Will it Compile or not

```
D:\Java_2018\FileHandling>javac FileIOException.java
FileIOException.java:6: error: unreported exception FileNotFoundException; must
be caught or declared to be thrown
    FileInputStream fis = new FileInputStream("abc.txt");
                          ^
FileIOException.java:7: error: unreported exception FileNotFoundException; must
be caught or declared to be thrown
    FileOutputStream fos = new FileOutputStream("xyz.txt");
                           ^
FileIOException.java:9: error: unreported exception IOException; must be caught
or declared to be thrown
    while((c=fis.read())!=-1)
                  ^
FileIOException.java:11: error: unreported exception IOException; must be caught
 or declared to be thrown
      fos.write(c);
          ^
FileIOException.java:13: error: unreported exception IOException; must be caught
 or declared to be thrown
    fis.close();
        ^
FileIOException.java:14: error: unreported exception IOException; must be caught
 or declared to be thrown
    fos.close();
        ^
6 errors
```

javap java.io.FileInputStream

```
D:\Java_2018\FileHandling>javap java.io.FileInputStream
Compiled from "FileInputStream.java"
public class java.io.FileInputStream extends java.io.InputStream {
  public java.io.FileInputStream(java.lang.String) throws java.io.FileNotFoundEx
ception;
  public java.io.FileInputStream(java.io.File) throws java.io.FileNotFoundExcept
ion;
  public java.io.FileInputStream(java.io.FileDescriptor);
  public int read() throws java.io.IOException;
  public int read(byte[]) throws java.io.IOException;
  public int read(byte[], int, int) throws java.io.IOException;
  public native long skip(long) throws java.io.IOException;
  public native int available() throws java.io.IOException;
  public void close() throws java.io.IOException;
  public final java.io.FileDescriptor getFD() throws java.io.IOException;
  public java.nio.channels.FileChannel getChannel();
  protected void finalize() throws java.io.IOException;
  static void access$000(java.io.FileInputStream) throws java.io.IOException;
  static {};
}
```

Handle the exception

```
import java.io.*;
class  FileIOExample
{
        public static void main(String[] args) throws FileNotFoundException, IOException
        {
                FileInputStream fis = new FileInputStream("abc.txt");
                FileOutputStream fos = new FileOutputStream("xyz.txt");
                int c;
                while((c=fis.read())!=-1)
                {
                        fos.write(c);
                }
                fis.close();
                fos.close();
        }
}
```

Output

- Make sure that two file must be created
- Data will be copied successfully

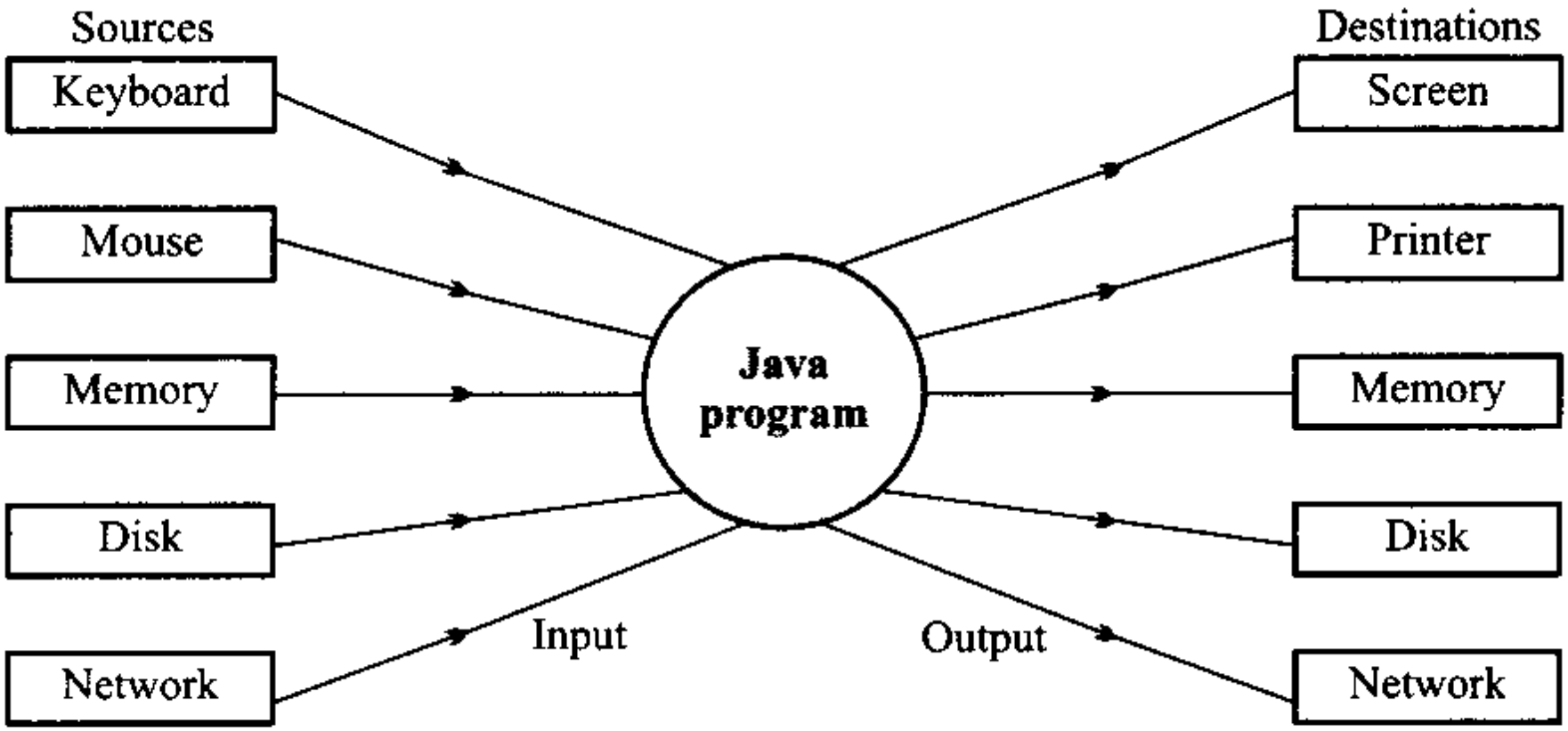
What will be the output here?

```
import java.io.*;
class  FileIOExample
{
        public static void main(String[] args) throws FileNotFoundException, IOException
        {
                FileInputStream fis = new FileInputStream("abc.txt");
                FileOutputStream fos = new FileOutputStream("xyz.txt");
                int c;
                while((c=fis.read())!=-1)
                {
                        System.out.println(c);
                        fos.write(c);
                }
                fis.close();
                fos.close();
        }
}
```

Output

```
D:\Java_2018\FileHandling>javac FileIOException.java

D:\Java_2018\FileHandling>java FileIOExample
97
98
99
100
```

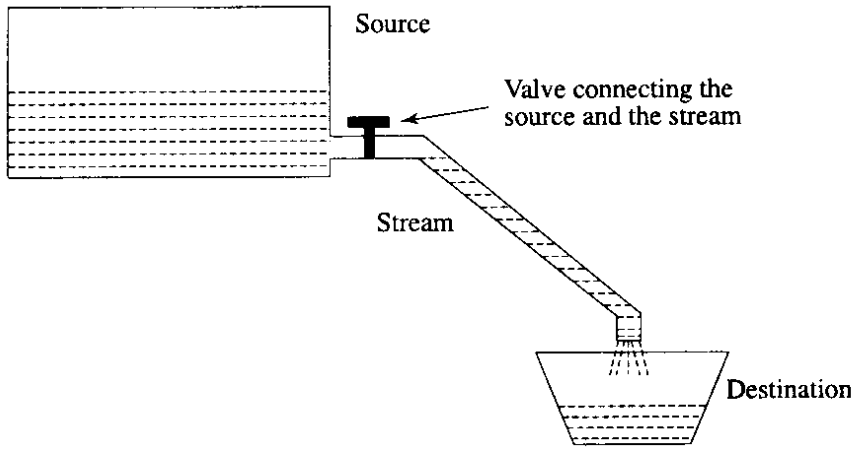
I/O and Data Movement



Relationship of Java program with I/O devices

Streams

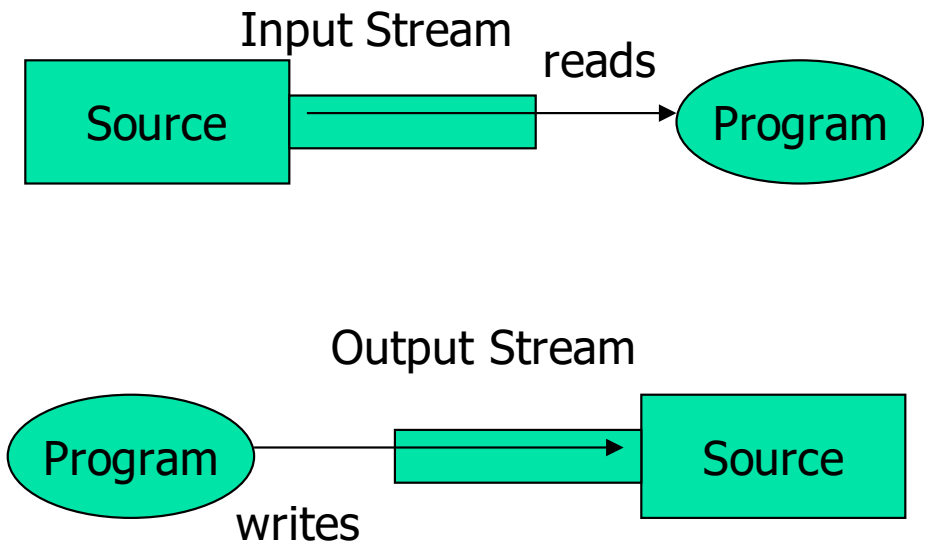
- Java Uses the concept of Streams to represent the ordered sequence of data, a common characteristic shared by all I/O devices.
- Streams presents a uniform, easy to use, object oriented interface between the program and I/O devices.
- A stream in Java is a path along which data flows (like a river or pipe along which water flows).



Conceptual view of a stream

Stream Types

- The concepts of sending data from one stream to another (like a pipe feeding into another pipe) has made streams powerful tool for file processing.
- Connecting streams can also act as filters.
- Streams are classified into two basic types:
  - Input Steam
  - Output Stream



Java Stream Classes

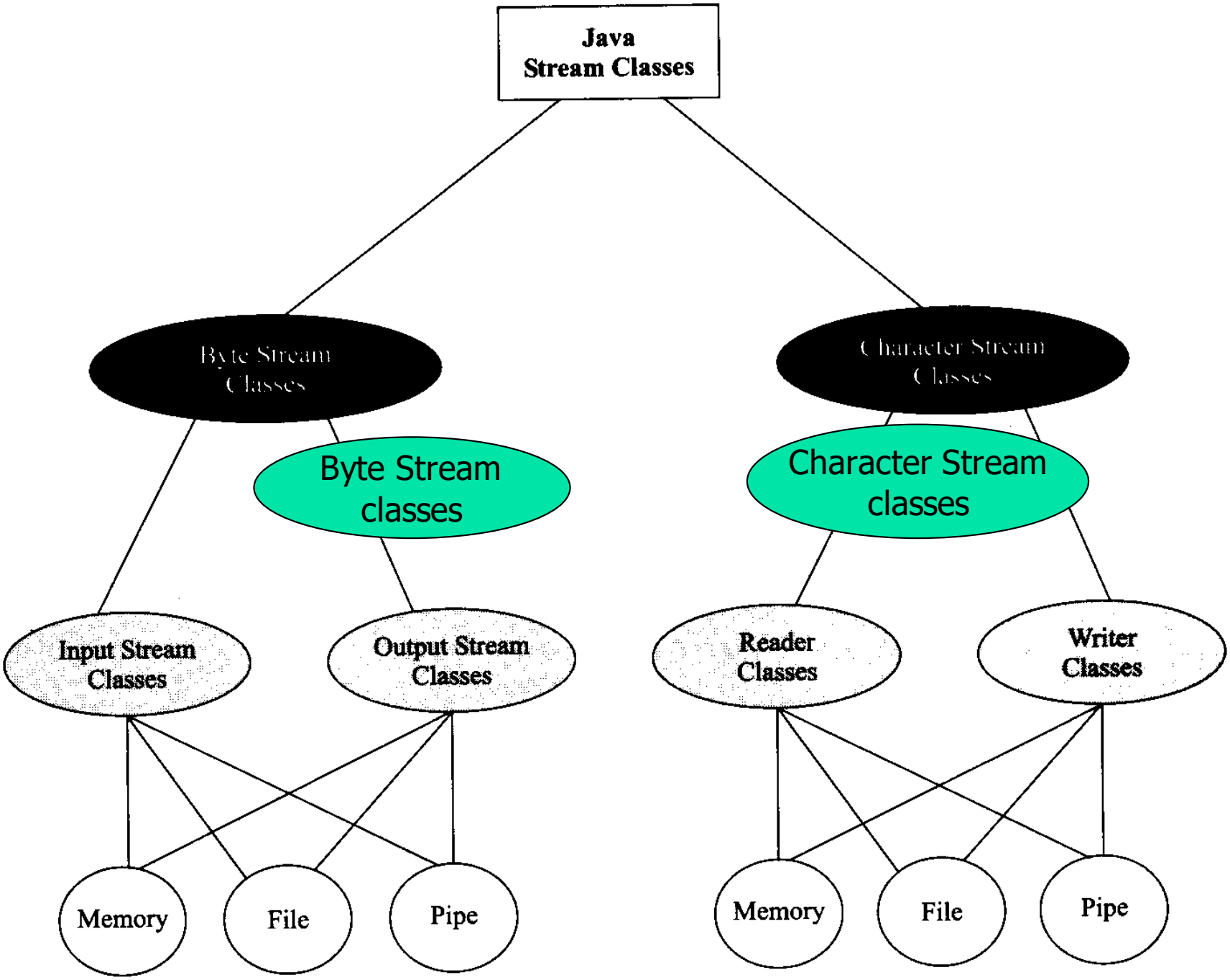
- Input/Output related classes are defined in java.io package.
- Input/Output in Java is defined in terms of streams.
- A *stream* is a sequence of data, of no particular length.
- Java classes can be categorised into two groups based on the data type one which they operate:
  - *Byte streams*
  - *Character Streams*

Streams

Byte Streams	Character streams
Operated on 8 bit (1 byte) data.	Operates on 16-bit (2 byte) unicode characters.
Input streams/Output streams	Readers/ Writers

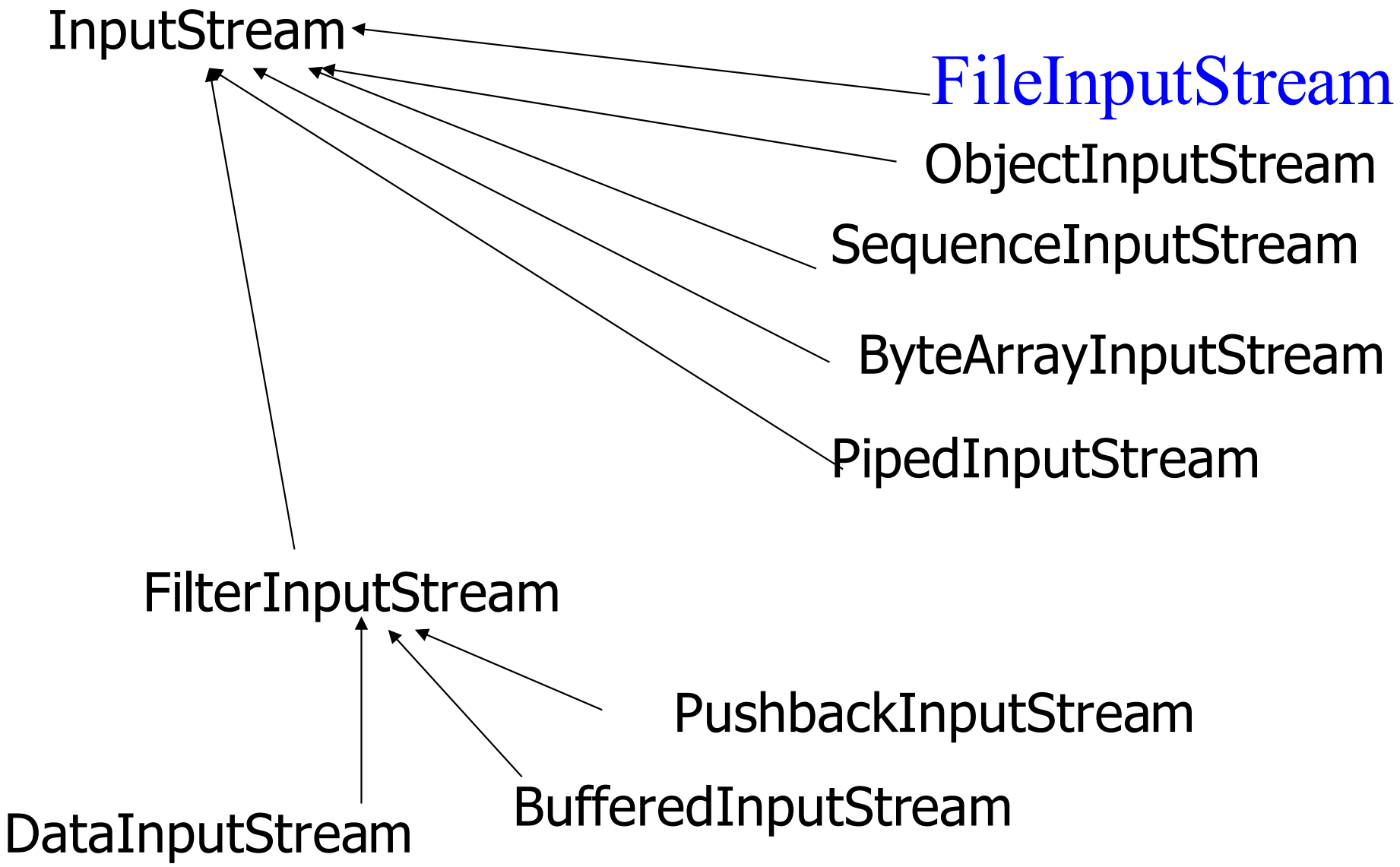
Difference Between Byte Stream and Character Stream

Byte Stream	Character Stream
Byte stream is used to perform input and output operations of 8-bit bytes.	Character stream is used to perform input and output operations of 16-bit Unicode.
It processes data byte by byte.	It processes data character by character.
Common classes for Byte stream are FileInputStream and FileOutputStream.	Common classes for Character streams are FileReader and FileWriter.
Example- Byte streams are used to read or write binary data.	Example- Character streams are used to read/write characters.



Classification of Java stream classes

Byte Input Streams



Java I/O – Using InputStreams

- Basic pattern for I/O programming is as follows:

```
Open a stream
While there's data to read
  Process the data
Close the stream
```

Java I/O – Using InputStreams

```
InputStream in = new
  FileInputStream("c:\\temp\\myfile.txt");
int b = in.read();

while (b != -1)
{
b = in.read();
}
in.close();
```

Java I/O – Using InputStreams

- But using buffering is more efficient, therefore we always nest our streams…

```
InputStream inner = new
      FileInputStream("c:\temp\myfile.txt");
InputStream in = new
      BufferedInputStream(inner);
int b = in.read();
while (b != -1)
{
  b = in.read();
}
in.close();
```

Byte Input Streams - operations

public abstract int read()	Reads a byte and returns as a integer 0-255
public int read(byte[] buf, int offset, int count)	Reads and stores the bytes in buf starting at offset. Count is the maximum read.
public int read(byte[] buf)	Same as previous offset=0 and length=buf.length()
public long skip(long count)	Skips count bytes.
public int available()	Returns the number of bytes that can be read.
public void close()	Closes stream

Byte Input Stream – example-2

- Count total number of bytes in the file

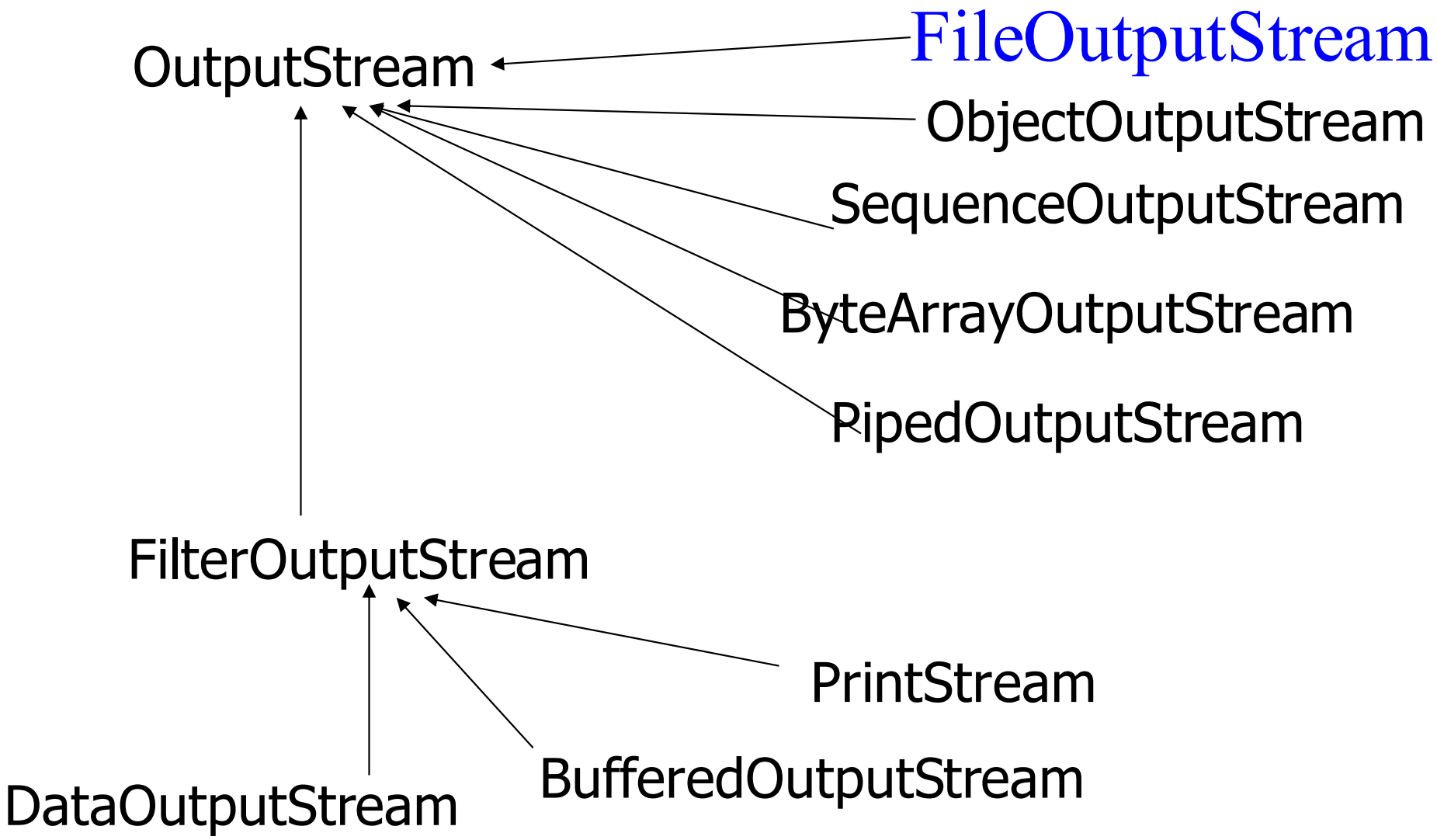
```
import java.io.*;

class CountBytes {
        public static void main(String[] args)
            throws FileNotFoundException, IOException
        {
                FileInputStream in;
                in  =  new FileInputStream("InFile.txt");

                int total = 0;
                while (in.read() != -1)
                        total++;
                System.out.println(total + " bytes");
        }
}
```

What happens if the file did not exist

- JVM throws exception and terminates the program since there is no exception handler defined.

```
Exception in thread "main" java.io.FileNotFoundException:
    FileIn.txt (No such file or directory)
        at java.io.FileInputStream.open(Native Method)
        at
    java.io.FileInputStream.<init>(FileInputStream.java:64)
        at CountBytes.main(CountBytes.java:12)
```

Byte Output Streams



Byte Output Streams - operations

public abstract void write(int b)	Write *b* as bytes.
public void write(byte[] buf, int offset, int count)	Write *count* bytes starting from *offset* in *buf.*
public void write(byte[] buf)	Same as previous *offset=0* and *count = buf.length()*
public void flush()	Flushes the stream.
public void close()	Closes stream

Byte Output Stream - example

- Read from standard in and write to standard out

```
import java.io.*;

class ReadWrite {
        public static void main(String[] args)
                throws IOException
        {
                int b;
                while (( b = System.in.read()) != -1)
                {
                        System.out.write(b);
                }

}}
```

Summary

- Streams provide uniform interface for managing I/O operations in Java irrespective of device types.
- Java supports classes for handling Input Steams and Output steams via java.io package.
- Exceptions supports handling of errors and their propagation during file operations.